Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation

Jose Renau James Tuck† Wei Liu† Luis Ceze† Karin Strauss† Josep Torrellas†

Dept. of Computer Engineering, University of California Santa Cruz
renau@soe.ucsc.edu

†Dept. of Computer Science, University of Illinois at Urbana-Champaign
{jtuck,liuwei,luisceze,kstrauss,torrellas}@cs.uiuc.edu

Abstract

Chip Multiprocessors (CMPs) are flexible, high-frequency platforms on which to support Thread-Level Speculation (TLS). However, for TLS to deliver on its promise, CMPs must exploit multiple sources of speculative task-level parallelism, including any nesting levels of both subroutines and loop iterations. Unfortunately, these environments are hard to support in decentralized CMP hardware: since tasks are spawned out-of-order and unpredictably, maintaining key TLS basics such as task ordering and efficient resource allocation is challenging.

While the concept of out-of-order spawning is not new, this paper is the first to propose a set of microarchitectural mechanisms that, altogether, fundamentally enable fast TLS with out-of-order spawn in a CMP. Moreover, we develop a fully-automated TLS compiler for aggressive out-of-order spawn. With our mechanisms, a TLS CMP with 4 4-issue cores achieves an average speedup of 1.30 for *full* SpecInt 2000 applications; the corresponding speedup for in-order-only spawn is 1.04. Overall, our mechanisms unlock the potential of TLS for the toughest applications.

1 Introduction

Chip Multiprocessors (CMPs) with Thread-Level Speculation (TLS) are being proposed as flexible, high-frequency engines to extract the next level of parallelism from hard-to-analyze programs (e.g. [10, 11, 12, 14, 20, 21, 22, 23, 29]). Under TLS, irregular sequential codes are divided into tasks that are executed in parallel, optimistically assuming that sequential semantics will not be violated. As the tasks run, the architecture tracks their control flow and data accesses. If a cross-task dependence is violated, the offending tasks are destroyed (*squashed*). Then, a repair action is initiated and the offending tasks are re-executed.

While these architectures have shown good potential, often due to sophisticated compiler support [2, 5, 13, 24, 25, 28], the speedups obtained for non-numerical applications have typically been modest. For example, for full SpecInt 2000 applications, the geometric mean speedups are 1.05 [28]. Part of the reason is that most designs have typically focused (often implicitly) on limited types of task structures: iterations from a single loop level (e.g. [4, 12, 28]); the code that follows (i.e., the continuation of) calls to subroutines that do not spawn other tasks (e.g. [3]); or some execution paths out of the current task (e.g. [25]). In the cases mentioned, *correct* tasks are spawned *in-order*, namely in the same order as they would execute sequentially[1]. While exploiting only these task structures may simplify the CMP hardware, it cripples its potential.

High-level performance evaluation studies have pointed out that there is a sizable amount of other parallelism available [15, 16, 26, 27]. One could execute in parallel all subroutines and their continuations irrespective of their nesting, and iterations from multiple loop levels in a nest. If this additional parallelism is harvested, the speedups are predicted to be significantly higher.

In practice, exploiting these additional sources of parallelism requires supporting *out-of-order* task spawning. For example, consider nested subroutines. When a task finds a subroutine call, it spawns a more speculative task to execute the continuation, while it proceeds to execute the subroutine. Inside the subroutine, the task can then find other subroutine calls, therefore spawning speculative tasks that are less speculative (i.e., less ahead in a sequential execution) than the one spawned first. The same occurs for nested loops, and for combinations of loop and subroutine nesting.

With out-of-order spawning, the application offers unpredictable shapes of parallelism that are hard to manage by TLS at run time. Specifically, how do we manage task ordering, which is required to identify violations and to ensure correct commit and squash? How do we balance resource allocation between highly-speculative tasks that have been running for a long time, and less speculative tasks that have just been spawned? To address these challenges with *minimal overhead* in a CMP, we need special microarchitecture.

The concept of out-of-order spawn is not new. In fact, there is a lot of related work in this area, which we detail in Section 9. However, no previous work has proposed a set of implementable microarchitectural mechanisms that, altogether, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. This paper is the first to do it. We view it as our main contribution.

Our simple mechanisms address the two main challenges posed by out-of-order spawning: correct and efficient task ordering and resource allocation. Task ordering is enabled with *Splitting Timestamp Intervals* for low-overhead order management, and the *Immediate Successor List* for efficient task commit and squash. Efficient resource allocation is enabled with *Dynamic Task Merging*, which directs speculative parallelism to the most beneficial code sections.

We have developed a complete, fully-automated TLS compiler for aggressive out-of-order spawn. With our mechanisms, a TLS CMP with 4 4-issue cores delivers an average speedup of 1.30 for *full* SpecInt 2000 applications; without out-of-order spawn, we obtain an average speedup of 1.04, in line with past

[1] Correct tasks do not include those that are in wrong branch paths.

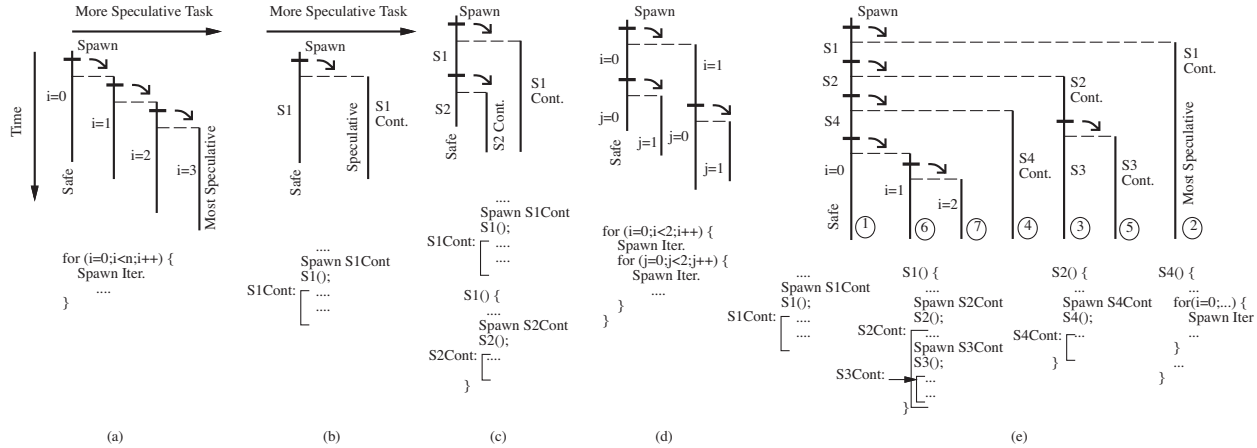

Figure 1: Example task trees. In the figure, Cont and Iter denote continuation and iteration.

TLS CMP work on the same codes (e.g., 1.05 in [28]). Overall, our mechanisms unlock the potential of TLS for the toughest applications, namely irregular integer codes.

This paper is organized as follows: Section 2 gives background on out-of-order spawning and why is needed; Sections 3 and 4 present our microarchitecture; Section 5 describes our compiler; Section 6 addresses complexity issues; Sections 7 and 8 present our evaluation; and Section 9 discusses related work.

2 Background: Out-of-Order Spawn

In most proposed TLS systems, tasks are formed with iterations from a single loop level (e.g., [4, 12, 28]), the continuation of calls to subroutines that do not spawn other tasks (e.g., [3]), or some execution paths out of the current task (e.g., [25]). In these proposals, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution of the program, rather than in the wrong path of a branch. As a result, *correct* tasks are spawned *in-order*, namely, in the same order as in sequential execution.

Figures 1-(a) and (b) show examples. Figure 1-(a) shows the task tree when parallelizing a loop. Each task spawns the next iteration. In the figure, the leftmost task is safe (or non-speculative); the more a task is to the right, the more speculative it is. Figure 1-(b) shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while a more speculative task is spawned to execute the continuation.

There is consensus that for TLS to deliver on its promise, it has to exploit more parallelism. Several high-level performance evaluation studies [15, 16, 26, 27], typically simulating simplified architectures, have pointed to the need to support nested subroutines and loop iterations.

Figures 1-(c) and (d) show the two cases. In Figure 1-(c), the safe task first spawns a task for the continuation of subroutine *S1*. Then, it enters *S1*, spawns a new task for the continuation of *S2*, and executes *S2* until its end. In Figure 1-(d), the safe task executes outer iteration 0. As it executes, it spawns outer iteration 1, enters the inner loop to execute inner iteration 0, and spawns inner iteration 1. When it completes inner iteration 0, it ends.

With these two task choices, an individual task can spawn multiple correct tasks. If so, correct tasks are spawned in strict reverse order compared to sequential execution. For example, in Figures 1-(c) and (d), the safe task spawns two correct tasks, and does so out of order, most speculative first. Figure 1-(e) is a more complex example: the time-line for task creation proceeds from top to bottom (*1-2-3-4-5-6-7*), while sequential order is from left to right (*1-6-7-4-3-5-2*).

In this paper, to discuss out-of-order spawning, we give examples of tasks built out of any nesting of subroutines and loop iterations, as they are an obvious source of TLS parallelism. Our analysis also applies to any other task structure that maintains two conventions. First, if a task spawns multiple tasks, the compiler inserts the spawns in strict reverse task order (last task is spawned first, etc). Second, the spawned tasks are less speculative than any task that was more speculative than their parent. These conventions are followed to make the spawn structure like that of nested loops and subroutines. Intuitively, these conventions are unlikely to affect task selection much, while they simplify the microarchitecture.

Out-of-order spawning enables more task parallelism: two code sections that are far-off in sequential execution can be executed in parallel *before* some of their intervening code sections have *even been spawned*.

3 Novel Microarchitectural Mechanisms

We propose three novel and simple microarchitectural mechanisms to enable high-speed tasking with out-of-order spawn in a TLS CMP. These mechanisms support task ordering and efficient resource allocation in an environment that is statically *unpredictable* (due to out-of-order spawn) and *decentralized* (due to the CMP architecture). We enable task order management with *Splitting Timestamp Intervals* (Section 3.1) and the *Immediate Successor List* (Section 3.2). We enable efficient resource allocation with *Dynamic Task Merging* (Section 3.3). In the following, when we use the terms successor and predecessor task, we refer to sequential execution order.

3.1 Splitting Timestamp Intervals for Task Order Management

In any TLS system, tasks have a relative order, which they explicitly or implicitly embed in the CMP protocol messages they issue and the cached data they own. Such order is most obviously needed when two tasks communicate. For example, consider a task reading cached data produced by a second task. The relative order of the tasks is assessed, and the data is provided only if the former task is a successor of the latter. Similarly, consider an invalidation message from a task to data read by a second task. The task order is considered and, if the reader is a successor, a dependence violation is triggered.

Under in-order task spawn, recording task order is easy:

since tasks are created in order, it suffices to assign monotonically increasing timestamps to newer tasks. A parent gives to its child its timestamp plus one. With this support, tasks with higher timestamps are successors of those with lower ones.

Such an approach does not work when tasks are created out of order. Consequently, we propose to represent a task with a *Timestamp Interval*, given by a *Base* and a *Range* timestamp ($\{$B,R$\}$). On a task spawn, the parent splits its timestamp interval in two pieces: the higher-range subinterval is given to the child (since it is more speculative), while the lower-range subinterval is kept by the parent. With this support, protocol messages and cached data are directly (or indirectly) associated with the base timestamp. Specifically, when tasks communicate, the base timestamps of the two tasks are compared *exactly* as in the in-order case.

As an example, Figure 2-(a) shows a program with a call to subroutine *S1*, which in turn calls *S2*. Assume that we use three tasks: task *i* executes the non-speculative code, *j* executes the continuation of *S1*, and *k* executes the continuation of *S2*. The resulting task tree is shown in Figure 2-(b), while Figure 2-(c) shows the timestamp intervals of each task.

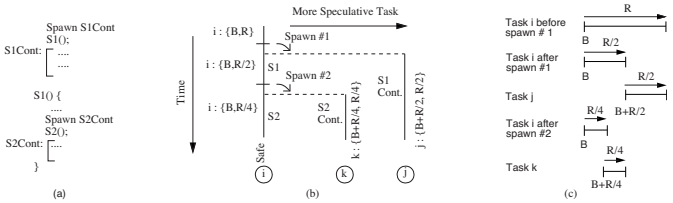

Figure 2: Changes in the base and range timestamps when tasks are spawned.

The example assumes that the initial interval for task *i* is $\{$B,R$\}$, and that intervals are partitioned in half. When *i* spawns *j*, *i* keeps $\{B,\frac{R}{2}\}$ and *j* obtains $\{B+\frac{R}{2},\frac{R}{2}\}$. When *i* later spawns *k*, *i* retains $\{B,\frac{R}{4}\}$ and *k* obtains $\{B+\frac{R}{4},\frac{R}{4}\}$. With this scheme, as we move from safe to most speculative task following sequential order (*i*, *k*, and *j*), we encounter increasing base timestamps (B, B+$\frac{R}{4}$, B+$\frac{R}{2}$).

In general, a simple approach is to give $\frac{1}{2}$ of the current interval to the child. However, since a task rarely spawns more than a few other tasks, it makes sense to give a larger fraction of the interval to the child. In addition, there are two cases where we can be more efficient. The first one is when the parent knows that the child will not spawn any task; in this case, the parent can give it a single timestamp. The second case is when the parent knows that this is its last child; in this case, the parent can keep a single timestamp. These efficiencies may be obtainable with information gathered by the compiler or hardware predictors.

Our scheme assigns no R to the most speculative task, which implicitly takes the maximum possible value representable by the R range (R_{max}). This allows the system to automatically and *dynamically expand* the range of used timestamps. Indeed, when the most speculative task spawns a child, it keeps the range $\{B,R_{max}\}$ for itself, and sets the base of the child to B+R_{max}. The child is now the new most speculative task, and implicitly takes the range $\{B+R_{max},R_{max}\}$.

Note that, it is possible that a program causes the timestamps to wrap around. In addition, in rare cases, a task may reach a point where it needs to spawn a child and its interval has size 1. These cases are discussed in Section 4.1.

Our scheme resembles Cleary *et al.*'s virtual sequences [7] without the implementation limitations (Section 9).

3.2 Immediate Successor List for Task Squash and Commit

In TLS, a task must be able to find its immediate successor very quickly, to perform the time-critical operations of commit and squash. Specifically, when the safe task commits, it passes the commit token to its immediate successor, which may be waiting for it to commit. As for squash, a task is squashed when it is found that it read data prematurely (data violation) or was spawned in the wrong branch path (control violation). In either case, the squashed task sends a kill signal to its immediate successor, which is propagated up to the most speculative task. This ensures that all possible side effects of the squashed task are erased.

Under in-order task spawn, it is easy for a task to find its immediate successor: the task spawned its immediate successor and it only needs to remember it. Alternatively, consecutively spawned tasks are often allocated on contiguous processors, making it trivial to identify the immediate successor. In other designs, a table with immediate successor information is used, which is easy to maintain because only one task can spawn at a time [6]. Overall, any scheme used is likely to be largely free of protocol races, as only one task spawns at a time.

Under out-of-order task spawn, identifying the immediate successor and all the more speculative tasks is not straightforward. For example, in Figure 1-(e), if task *7* is killed, it is not trivial for it to identify and kill tasks *4*, *3*, *5*, and *2*, which were created before and independently of *7*. Moreover, any solution has to be carefully crafted to avoid inducing races in the TLS protocol of the distributed CMP if multiple operations happen concurrently. Finally, since commit and squash are time-critical, we cannot use a solution based on repeated comparison of timestamps.

To support efficient and race-free commit and squash, we propose that the tasks dynamically link themselves in hardware in a list according to their sequential order. We call this list the *Immediate Successor* (IS) list. To build the IS list, we add a hardware pointer called the IS pointer to each task structure. We leverage the fact that, at the time of the spawn, (i) the child becomes the immediate successor of its parent, and (ii) the child inherits the parent's previous immediate successor. Consequently, on a spawn, the child receives the parent's IS pointer, and the parent sets its IS pointer to point to the child. In the example of Figure 1-(e), the IS list links *1* to *6*, *6* to *7*, *7* to *4*, and so on.

When a task kills all its successors, its IS is set to nil. Consequently, Task *2*'s IS pointer in Figure 1-(e) is nil.

With this support, when a task needs to pass the commit token, it uses the IS list. Moreover, when a squashed task needs to kill all its successors, it sends a kill signal with its own identity downstream the IS list. All successors are killed in turn. When the kill signal reaches a task with a nil IS, an acknowledgment is sent to the originating task, which sets its IS to nil. The result is very fast commit and squash. In addition, the TLS protocol implementation is simplified in a major way: even when multiple kill signals occur concurrently, since all signals are serialized along the same path, protocol races are minimized.

3.3 Dynamic Task Merging for Efficient Resource Allocation

In TLS systems, tasks compete for CMP resources such as CPUs or cache space. Under out-of-order task spawn, such competition is harder to manage. The reason is that highly-speculative tasks may hog resources and starve more critical

(less speculative or even safe) tasks that are spawned later. For example, in Figure 1-(e), when safe task *1* is about to spawn *6*, all the CPUs in the CMP may be in use by more speculative tasks *4*, *3*, *5*, and *2*.

To allocate chip resources efficiently, we propose a new CMP microarchitectural technique that we call *Dynamic Task Merging*. It consists of transparent, hardware-driven merging of two or more consecutive tasks at run time. In effect, it enables the CMP to prune some branches of the task tree based on dynamic load conditions.

Task merging increases execution efficiency in several ways. First, highly-speculative tasks can be merged, therefore freeing resources for more critical tasks. Second, with large, merged tasks, the spawn overhead is less noticeable, and both caches and branch predictors work better. Finally, given that the hardware can adjust the number of tasks at run time, the TLS compiler can be more aggressive at creating tasks, which may ultimately lead to higher performance.

Under task merging, a task skips the spawn instruction for a child, and its own task-end instruction. Figure 3 illustrates it. When task *1* finds the spawn for *3* (Chart a), it can either spawn (Chart b) or merge (Chart c). In the rest of this section, we discuss the microarchitecture support and the heuristics used for task merging. Some compiler issues are discussed in Section 5.2. Some related, but less flexible proposals are discussed in Section 9.

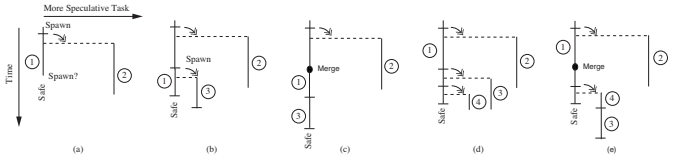

Figure 3: Examples of task merging.

3.3.1 Microarchitecture

A task initiates a merge by skipping a spawn instruction. After that, in the simplest case, the task will also have to skip the first task-end instruction that it finds, and finish only when it finds the second task-end. In general, if a task initiates *N* merge operations by skipping *N* spawns, it will have to also skip *N* task-ends and finish only when it finds the *N+1* one.

Consider now a task that skipped a spawn and later, because *load conditions have changed*, it wants to spawn a child. Since the child is more speculative than the parent, the parent *passes the responsibility* to complete the task merge to the child: the child will skip the first task-end that it finds and finish only at the second one. As for the parent, it simply finishes at the first task-end that it finds.

As an example, consider Figure 3-(d), which shows a new task tree without merging. In Figure 3-(e), we show the same tree except that task *1* merged with *3* and then spawned *4*. We see that task *4* is given the responsibility of completing the merge.

The microarchitecture needed to support task merge is a counter in each task structure called *Number of Ends to Skip* (NES). When a task skips a spawn, its NES is incremented. When a task finds a task-end instruction, its NES is checked. If it is non-zero, it is decremented and the end instruction is skipped. Otherwise, the task ends. Moreover, when a task spawns a child, the parent's NES is copied to the child's and is then cleared. The child now owns the merges.

When a task becomes the most speculative one (its IS pointer becomes nil), its NES ceases to matter — the task always skips any task-end instruction that it finds. This is the appropriate behavior for the most speculative task, which should not be stopped by end instructions. Indeed, the most speculative task may be the only task in the system, possibly the result of a task killing all its successors on a violation; if it were allowed to end on a task-end instruction, progress would stop. Aside from this case, the most speculative task handles the NES as usual. In particular, it sets its NES to zero on child spawn.

To reduce overhead, the NES is checked and modified in hardware. A software implementation is also feasible, although it can be shown to need 5-7 instructions on task spawn and end.

3.3.2 Heuristics

To understand our merge heuristics, note that we rely on squash information to reduce useless work. Specifically, if a given task has been squashed and restarted repeatedly due to violations, it is preempted and not allowed to get a CPU anymore. It remains stalled in one of the several on-chip task containers (Section 4.3) until it becomes safe. This policy prevents highly-speculative, frequently-squashed tasks from clogging CPUs. On the other hand, if a CPU is running, we will assume that it is doing useful work.

With this support, we use CPU usage to decide whether or not to perform task merge. Specifically, every time that a task finds a spawn instruction, it performs a merge if all CPUs are busy. In addition, every *NumMNext* merges, we skip one to prevent tasks from becoming so large that a squash would discard a lot of work.

4 Implementation Issues

This section discusses some related implementation aspects. Of those, only the first two are specific to out-of-order spawning.

4.1 Special Cases in Timestamp Intervals

There are two special cases when handling timestamp intervals. The first one is when the timestamps are about to wrap around. Our solution is to recycle old timestamps in chunks. For that, we divide the whole representable timestamp range into four chunks, based on the two most significant bits of B. When all the tasks with intervals in the lowest chunk (e.g., the 00 chunk) have committed, we recycle that chunk. This involves sending a reprogramming signal to the logic of the timestamp comparators so that timestamps in the recycled chunk are now the highest (i.e. 00, is more speculative than 11). Then, timestamps from that chunk become available for reassignment to newer tasks.

The reprogramming signal is automatically triggered when a task with an interval that straddles two chunks commits. With this approach, the tasks in the CMP can at most use $\frac{3}{4}$ of the whole timestamp range at a time. To see how many tasks can be concurrently supported, assume that B and R have *b* and *r* bits, respectively. If, in the worst case, each task has a single child, and the child is given the maximum timestamp range possible (2^r), the maximum number of tasks is then $\frac{3}{4} \times 2^{b-r}$. Consequently, if we want to support about 20 concurrent tasks, $b - r$ should be at least 5.

The second, infrequent case, is when a task wants to spawn a child and has no interval to assign. In this case, the task simply sends a kill signal down the IS list, as in a task squash (Section 3.2). This operation kills all successors, making the task the most speculative one. At that point, the task obtains R_{max} timestamps (Section 3.1). This operation, however, is very rare, thanks to our support for automatic dynamic times-

tamp expansion (Section 3.1) and timestamp wrap around. In our experiments, it rarely occurs at all.

4.2 Scheduling Tasks to CPUs

While all the tasks that have been spawned have their state loaded on on-chip task containers (Section 4.3), only as many tasks as CPUs can be running at a time. In practically all TLS proposals, tasks are scheduled strictly based on how speculative they are. Specifically, a less speculative task always preempts more speculative ones.

In practice, our experiments show that such a policy is an overkill, given our new task merging support. Consequently, we use a simpler policy: we assign a high priority to the safe task, and a fixed, low priority to all speculative tasks.

4.3 Other Aspects

All the other aspects of a TLS CMP largely remain the same as we move from an in-order to an out-of-order spawning framework. Each processor has a table of task containers, which keeps state for the tasks that are loaded on the processor. Of these tasks, only one is running at a time. Each container stores the start PC of the task, a pointer to a stack location with saved-register state, and a local ID associated to the task. The state saved in the stack is not read at the beginning of the task. Rather, it is read as registers are needed. As in many TLS systems, the ID is a short ID used to tag the cache lines accessed by the task. It acts as a form of indirection [21] that avoids the need to tag the cache lines with the B timestamp of the task. For the out-of-order spawn framework, a task container also contains the B, R, IS pointer and NES.

Our CMP uses a TLS coherence protocol with lazy commit inspired in [18] to detect memory-based data dependence violations. No special support for register communication between CMP cores is present. Cache lines with speculative state cannot be evicted. If the space taken by one such line is needed, the owner speculative task is squashed. Context switches and exceptions also cause squashes.

5 Compilation for Out-of-Order Spawn

We have developed a fully automated TLS compiler that generates in-order and out-of-order tasking out of sequential, integer applications. The compiler adds several passes to a development branch of gcc 3.5. The branch uses a static single assignment tree as the high-level intermediate representation [9]. Building on this software allows us to leverage a complete compiler infrastructure. Also, working at this high level is better than using a low-level representation such as RTL: we have better information and it is easier to perform pointer and dataflow analysis. At the same time, our transformations are much less likely to be affected by unwanted compiler optimizations than if we were working at the source-code level.

5.1 Task Generation and Hoisting

Our compiler uses the following modules as potential tasks for both the in-order and out-of-order environments: subroutines from any nesting level, their continuations, and loop iterations from multiple loops in a nest. All subroutines are potentially chosen unless they are very small. Recursion is handled seamlessly. In loop nests, the compiler makes decisions based on loop iteration size, which has to be larger than a certain minimum.

The actual tasks that make it to the final binary are different in the in-order and out-of-order environments. The out-of-order pass can select all the tasks mentioned, subject to some pruning heuristics, without worrying about the number of children per task. The in-order pass has to be more careful, since a task can only have a single child. Consequently, the in-order pass has an initial step where it analyzes all the files in the program and generates a complete task call graph. Then, using heuristics about task size and overheads, it eliminates tasks from the graph until each task only has a single child. We trust the quality of our heuristics based on the fact that the resulting in-order TLS code obtains speedups comparable to previous work [28].

Once the tasks and their parents are selected, the compiler inserts spawn instructions, and tries to hoist them to boost parallelism. A spawn is hoisted as far up as we can, as long as the new position is execution equivalent with the start of the task to spawn. We do not hoist above statements that can cause data or control dependence violations. Under out-of-order spawn, we make sure that the tasks are spawned in reverse order. Under in-order spawn, a spawn cannot be hoisted above the caller task.

A final task clean-up pass looks for spawns that were hoisted only a handful of instructions. In this case, the spawn is eliminated, and the two corresponding tasks integrated into one. This reduces overheads.

As an example, Figure 4 shows how the compiler generates out-of-order tasks out of a subroutine and its continuation. Chart (a) shows the dynamic execution in and out of the subroutine. The compiler marks the subroutine and continuation as tasks, and inserts two spawn instructions in the caller (Chart (b)). Then, it hoists the spawn for the continuation (Chart (c)) and subroutine (Chart (d)). In Chart (e), the clean-up pass eliminates the subroutine spawn because it had little hoisting.

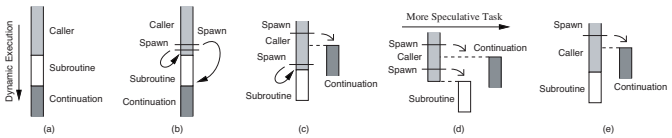

Figure 4: Generating tasks out of a subroutine and its continuation.

5.2 Task Merging

With task merging, as a task completes its code, it goes on executing the code of its immediate successor. This means that the task must have a way of obtaining the live-in register values for its continuation code. With our compiler, this is possible: all register values changed by a task that may be used by successors are stored in memory when the task finishes. Moreover, all the live-ins of a task are read from memory. Consequently, as a task merges with its successor, it automatically reads from memory the live-ins of the successor.

5.3 Offline Profiling Support

The compilation process for in- and out-of-order tasking includes running a simple profiler. The profiler takes the TLS executable and identifies those task spawn points that should be removed because they are likely to induce harmful squashes according to our models. The profiler returns the list of such spawns to the compiler. Then, the compiler generates the final TLS executable by removing these spawns

The profiler takes a few minutes to run. It executes the binaries sequentially, using the Train data set of the SpecInt codes. As the profiler executes a task, it records the variables written. As it executes tasks that would be spawned earlier, it compares the addresses read against those written by predecessor tasks. With this, it estimates potential violations. The profiler also models a cache to estimate the number of misses in the ma-

chine's L2. For speed, the cache model is timeless.

The profiler identifies those spawns where the ratio of squashes per task commit is higher than R_{squash}. For each of those spawns, it estimates the performance benefit that a task squash brings. Some benefit comes from the data prefetching provided by cache misses recorded before the task is squashed ($M_{squashed}$). Other benefit comes from true overlap of the instructions in the task with other tasks, as the task is re-executed after the squash ($I_{overlap}$). With these measurements, the profiler requests spawn removal if $T_I \times I_{overlap} + T_0 \times M_{squashed}$ is less than a threshold T_{perf}. In the formula, T_0 is the estimated stall per L2 miss, and T_I is the estimated execution time per instruction.

6 Complexity of Supporting Out-of-Order Spawn

Adding support for out-of-order spawn to a TLS CMP that already supports in-order spawn does not introduce much hardware complexity. The reason is that our mechanisms only add modest-sized, decentralized logic structures. No core-to-core interconnections are added. Finally, out-of-order spawn simplifies our TLS compiler. In the following, we discuss these issues.

Splitting Timestamp Intervals.

The memory hierarchy and protocol of the TLS CMP is oblivious to the fact that tasks use timestamp intervals rather than single timestamps. The reason is that all operations use B (the base timestamp), which remains unchanged for a task's lifetime, exactly like under in-order spawn. Intervals are only operated on at a task spawn, where they are split in hardware. This operation is very simple, as it involves one shift, two integer additions and two selection operations (multiplexers), as shown in Figure 5(a). Moreover, it is performed *locally* in each processor.

The case of comparator reprogramming because of timestamp wrap around is too infrequent to deserve any complicated implementation. The comparator in each cache is prompted to change how it orders the values of the two most-significant bits of timestamps, illustrated in Figure 5(b). When the comparators are being reprogrammed, tasks temporarily use merge rather than spawn.

Finally, if a task wants to spawn and has no interval to assign, it squashes its successors. This operation is already present in in-order spawn systems.

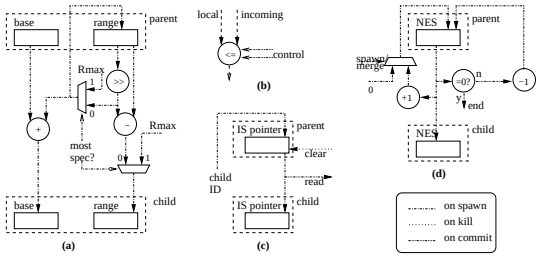

Figure 5: Possible implementation of out-of-order mechanisms.

Immediate Successor List.

The IS list is a decentralized structure that is maintained with simple hardware operations performed locally in each processor. Specifically, on task spawn, the parent's IS pointer register is read and written, while the child's is written. On a task commit or kill, the IS pointer is read and maybe cleared. There is no global, centralized operation, as shown in Figure 5(c).

Dynamic Task Merging.

The NES register is potentially read and written at task-spawn and task-end instructions. These are simple operations that are performed locally in each processor. We support them in hardware, as shown in Figure 5(d), although they can be supported with 5-7 instructions.

To decide on task merging at a spawn instruction, a CPU uses information about the state of the other CPUs (busy or not). This information does not need to be cycle accurate, since it is only used as a heuristic. Consequently, CPUs use normal links to regularly inform other CPUs of their busy or idle state.

Out-of-Order Spawn Simplifies our TLS Compiler.

Hardware support for out-of-order spawn simplifies our TLS compiler because it eliminates the checks necessary to ensure that tasks are only spawned in order. To guarantee the latter, our in-order-spawn pass has to identify the tasks that may be spawned at run time, and ensure that each task has at most one child. It does so with inter-procedural analysis. In contrast, our out-of-order spawn pass has many fewer checks to make. In particular, it just performs intra-procedural analysis.

Finally, with our dynamic task merging, a TLS compiler does not need to be as careful in creating load-balanced tasks; the hardware will prune the excess of tasks through merging.

7 Evaluation Methodology

To evaluate TLS with out-of-order spawn, we use execution-driven simulations with detailed models of out-of-order superscalars and memory systems. The proposed architecture is a four-core CMP with TLS support called *TLS4*. Each processor in *TLS4* is 4 issue and has a private L1 cache that can hold speculative state. The chip also has a shared L2 that only holds safe data. The 4 L1 caches and the L2 are connected through a switch that can support up to 2 concurrent connections. The CMP uses a TLS coherence protocol with lazy commit inspired in [18] to detect memory-based data dependence violations. To keep the hardware simple, there is no special support for register communication between CMP cores. For a similar reason, there is no dependence predictor to alleviate the impact of dependence violations.

In our experiments, we also model three other chips. Two of them are architectures built out of the 4-issue cores in *TLS4*: *4issue* and *TLS2*. *4issue* is a chip with a single core, one L1, and one L2. *TLS2* is like *TLS4* but with only 2 cores; for simplicity, all other parameters are the same as in *TLS4*. *TLS4* and *TLS2* use the microarchitecture introduced in Sections 3 and 4. Finally, we also model a chip with a single 6-issue processor, one L1, and one L2. We call this architecture *6issue*. The parameters used for the architectures are shown in Table 1.

To compare the TLS chips (*TLS4* and *TLS2*) to the non-TLS ones (*4issue* and *6issue*), we make the following assumptions. First, we increase the L1 access time in the TLS chips one extra cycle to 3 cycles. We do it to account for any time overhead that TLS may add. We think that such overhead is correctly modeled with 1 cycle. To see why, recall that TLS extends a line's tag with (i) a small task ID (6 bits in our case) and (ii) two read/write access bits per word in the line. The task ID bits can be considered part of the address tag, as a hit requires address and ID match. When a processor accesses its L1, it sends the address plus the ID of the running task. ID comparison occurs in parallel with tag comparison, and adds no extra delay. Moreover, in our protocol, the read/write access bits are not checked before providing the data to the processor; they may be updated after that. Consequently, the time overhead added

Processor Parameters	*TLS4 [TLS2]*	*4issue [6issue]*
Cores/chip	4 [2]	1
Running tasks/core	1	1
TLS hardware?	Yes	No
Frequency, technology	5 GHz, 70 nm	5 GHz, 70 nm
Fetch, issue, retire width	4, 4, 4	4, 4, 4 [6, 6, 6]
ROB, I-window size	152, 80	152, 80 [204, 104]
LD, ST queue	54, 46	54, 46 [66, 54]
Mem, int, fp units	2, 3, 1	2, 3, 1 [2, 5, 2]
Branch predictor:		
Penalty	14 cycles	14 cycles
BTB	2 K, 2 way	2 K, 2 way
global gshare(11) entries	16 K	16 K
local 2 bit entries	16 K	16 K
L1 cache:		
size, assoc, line	16 KB, 4, 64 B	16 KB, 4, 64 B
OC, RT	1, **3**	1, **2**

Tasking Parameters	Common Memory System
Task containers/processor: 8	L2 cache
B, R timestamp size: 32, 22 bits	size, assoc, line : 1 MB, 8, 64 B
NumMNext : 8	OC, RT : 1, 11
Latencies in cycles (min):	
From spawn to new thread: 14	Memory
From violation to full kill notification: 20	bandwidth : 10 GB/s
Drain proc pipeline: 14	RT : 500 cycles
Fraction of interval given to child: 3/4	
R_{squash}: 0.8, T_0: 200 cycles,	RT to neighbor's L1 (min) : 8 cycles
T_I: 1 cycle, T_{perf}: 100 cycles	

Table 1: Architectures considered. In the table, OC and RT stand for occupancy and minimum-latency round trip from the processor, respectively. All cycle counts are in processor cycles. In our comparison, we use the same processor frequency for all architectures.

by TLS is very small.

Secondly, we set all the L1 caches to the same size, to ensure that they all have the same cycle time. Although, as a result, the TLS chips have 4 or 2 times as much L1 as the non-TLS chips, increasing the size of the L1 in the non-TLS chips could hurt their performance because the cycle time would increase. In reality, the miss rate of each L1 in the TLS chips is higher than in the L1 of the non-TLS chips due to TLS effects.

Finally, we assume the same processor frequency in all chips. In a real implementation, the frequency of *6issue* would probably be lower than the that of *TLS4*. However, this paper makes its point without considering frequency effects.

We drive our simulated architectures with the SpecInt 2000 applications running the Reference data set. We run all the SpecInt 2000 codes except *eon* (we cannot compile because it is in C++) and *gcc* and *perlbmk* (our compiler infrastructure does not compile them). We compare the SpecInt binaries of Table 2: unmodified binaries (*BaseApp*), TLS with in-order spawning (*InOrder*), and TLS with out-of-order spawning (*OutOrder*). All the binaries in Table 2 are compiled with the same compiler options; *BaseApp* is compiled with our TLS passes disabled.

Name	TLS?	Description of Binary
BaseApp	N	Out-of-the-box, sequential version compiled with *O2*. No TLS instrumentation.
OutOrder	Y	Out-of-order task spawning. Spawns: subroutines, subroutine continuations, and loop iterations
InOrder	Y	In-order task spawning. Same tasks as *OutOrder*. However, it uses interprocedural analysis pass to eliminate tasks that (may) violate the in-order spawning requirement.

Table 2: Versions of the SpecInt 2000 binaries executed.

These binaries are different. The TLS passes re-arrange code into tasks and add spawn and commit instructions. Such transformations obfuscate some conventional compiler optimizations, sometimes rendering them less effective. Consequently, to accurately compare the performance of the different binaries, we cannot simply time a fixed number of instructions. Instead, we insert "simulation markers" in the code, and simulate for a given number of them. After skipping the initialization (several billion instructions), we execute up to a certain number of markers for all binaries, so that *BaseApp* graduates more than 750 million instructions.

8 Evaluation

8.1 Overall Execution Speedups

To evaluate out-of-order spawn for TLS, we compare the execution time of the *InOrder* and *OutOrder* binaries running on the *TLS4* architecture. For comparison purposes, we also measure the execution times of the *BaseApp* binary running on the *4issue* and *6issue* architectures. The comparisons to *4issue* and *6issue* show the speedup of TLS relative to a single processor of the same width and a wider one, respectively, always under the same frequency. Finally, we also run *OutOrder* on *TLS2*, to assess the effect of the number of processors in the CMP.

Figure 6 shows the speedups of the different binary-architecture combinations relative to *BaseApp* running on *4issue*. The figure shows speedups for each application and the *geometric* mean. On top of some bars, we show the speedups. The dots on some bars will be discussed later. The average IPC of each application for *4issue* and *TLS4OutOrder* is shown in Columns 2 and 3 of Table 3, respectively.

Compare first *TLS4OutOrder* to *4issue*. For every single application, TLS execution is faster. The speedups are always over 1.08, and reach about 2.4 for *mcf*. We will see that the *mcf* speedups are mostly due to prefetching. The geometric mean is 1.30 including *mcf*, and 1.20 not including *mcf*. For the two-core TLS CMP (*TLS2OutOrder*), the geometric mean of the speedup is 1.20. These results make TLS an attractive feature, especially given that these speedups are obtained with a *fully-automated* TLS compiler, on a *decentralized* CMP architecture and, importantly, on *full* SpecInt applications.

Note that the IPC numbers in Table 3 do not exactly correlate with the relative height of the *4issue* and *TLS4OutOrder* bars because the binaries running on the two platforms differ.

Since the contribution of this paper is support for out-of-order spawning, we compare *TLS4OutOrder* to *TLS4InOrder*. The bars show that *TLS4InOrder* is much slower in all applications. *TLS4InOrder* only obtains a geometric mean speedup of 1.04 over *4issue*. The magnitude of this figure is in line with previous compiler-driven TLS evaluations of SpecInt2000 codes on CMPs [28], if we weight the speedups reported by the coverage of the regions that were sped up.

We conclude, therefore, that out-of-order spawn is a key enabler to boost TLS speedups. The gains come from being able to exploit the additional sources of parallelism.

Finally, we compare *TLS4OutOrder* to *6issue*. The bars show that, except for *bzip2* and *gzip*, the TLS architecture outperforms the wider superscalar. Looking at the geometric mean, we see that *TLS4OutOrder*'s speedup is 20% higher. Therefore, a TLS CMP architecture compares favorably against a wider superscalar for SpecInt, even assuming that the wider superscalar cycles at no lower frequency. This is significant, given that the CMP has a natural advantage on truly parallel codes, such as many numerical applications.

8.2 Understanding TLS Speedups

To understand the speedups of *TLS4OutOrder*, we break down the execution time of *4issue* and *TLS4OutOrder* into the product of committed instructions times average CPI. In the formula, CPI_{TLS} corresponds to the combined CPI of all the

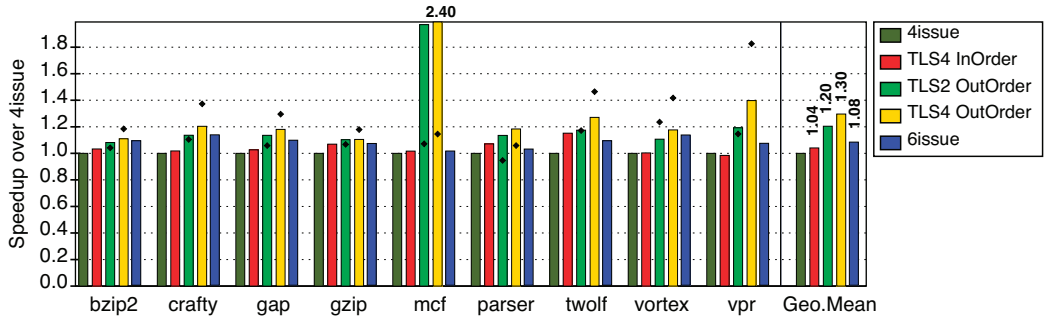

Figure 6: Speedups of different binary-architecture combinations relative to *BaseApp* running on *4issue*. The figure also shows the geometric mean. The TLS results are obtained with a fully-automated TLS compiler on full SpecInt applications.

cores in the chip.

$$Speedup_{TLS} = \frac{T_{4issue}}{T_{TLS}} = \frac{I_{4issue} \times CPI_{4issue}}{I_{TLS} \times CPI_{TLS}}$$

As we go from *4issue* to *TLS4OutOrder*, the number of committed instructions in a program increases. The reasons are the additional spawn, commit and memory instructions, and the lower effectiveness of conventional compiler optimizations (Section 7). Therefore, we define an instruction bloat factor $f_{bloat} = \frac{I_{TLS}}{I_{4issue}}$.

We can put CPI_{TLS} as a function of the average CPI per core, which we call $CPI_{TLScore}$. For that, we need to measure the time each CPU is busy executing instructions (t_i) and add it up across all the CPUs in the TLS chip as follows:

$$I_{TLS} = \frac{T_{TLS}}{CPI_{TLS}} = \frac{\sum_{i=1}^{numcores} t_i}{CPI_{TLScore}}$$

Intuitively, if no two CPUs are busy at the same time, there is no parallelism, and the two CPIs are the same. If all 4 CPUs completely overlap their busy time, parallelism is 4, and $CPI_{TLScore}$ is 4 times CPI_{TLS}. We express parallelism as:

$$f_{parallel} = \frac{\sum_{i=1}^{numcores} t_i}{T_{TLS}} = \frac{CPI_{TLScore}}{CPI_{TLS}}$$

Consequently, the TLS speedup above is:

$$Speedup_{TLS} = \frac{I_{4issue} \times CPI_{4issue}}{I_{TLS} \times CPI_{TLS}} = \frac{f_{parallel} \times CPI_{4issue}}{f_{bloat} \times CPI_{TLScore}}$$

Table 3 shows the values of some of these parameters for *TLS4OutOrder* running each of the applications. Column 4 shows the instruction bloat factor f_{bloat}. Its average value is 1.15, which indicates that TLS execution increases the dynamic instruction count significantly. This effect hurts TLS speedups. Column 5 shows the parallelism factor $f_{parallel}$, which helps TLS speedups. On average, its value is 1.53. $f_{parallel}$ is small because of the limited parallelism present in SpecInt codes. Note that $f_{parallel}$ reports the average number of CPUs that are busy at a given time executing tasks that will not be squashed. In reality, a higher number of CPUs is busy, but some of them execute tasks that will eventually be squashed. The true number of busy CPUs is shown in Column 6. Its average value is 1.96. We can see, therefore, that task squashing is not negligible. In fact, Column 7 shows the fraction of busy cycles that correspond to squashed tasks. On average, such number is 20.5%. Overall, *TLS4OutOrder* wastes many cycles to squashed tasks, which also limits its speedups[2].

We can now go back to the $Speedup_{TLS}$ equation and assume that $CPI_{4issue} = CPI_{TLScore}$. In this case, the TLS speedups would be given by $\frac{f_{parallel}}{f_{bloat}}$. We have computed this ratio and shown it as dots in Figure 6 for *TLS2OutOrder* and *TLS4OutOrder*.

If these dots are not equal to the real speedups, it is because $CPI_{4issue} \neq CPI_{TLScore}$. In particular, if a dot is lower than the TLS bar (e.g. in parser), it means $CPI_{4issue} > CPI_{TLScore}$. This is largely due to *prefetching* effects. In particular, tasks that eventually get squashed bring data and instructions into the caches, which are later reused by other tasks. If, instead, the dot is higher than the TLS bar (e.g. *vortex*), it means $CPI_{4issue} < CPI_{TLScore}$. In this case, TLS execution is largely impaired by the higher average memory latency induced by cache-coherence invalidations, higher instruction cache miss rate, and slower cache hierarchy speed. It is also hurt by lower branch predictor accuracy due to code partitioning. We call these effects TLS overheads.

Figure 6 shows that in *TLS2OutOrder*, the prefetching effect typically dominates (most of the dots are lower than the TLS bars). It often adds a net 5-10% to the potential speedup from parallelism, represented by the dots. However, as we add more cores to the chip (*TLS4OutOrder*), the TLS overheads dominate, and often the potential speedup from parallelism is higher than the real speedup by a net 10-30%. The obvious exceptions are *mcf* and *parser*, where prefetching always dominates, and *vortex*, where the TLS overheads dominate. *mcf* benefits significantly from prefetching into the L2. Its L2 miss rate decreases by 27% from *4issue* to *TLS4OutOrder*. *vortex* hurts from higher data and instruction L1 miss rates.

Overall, we conclude that our full SpecInt speedups are a combination of several factors. Our TLS machinery is frequently able to overlap execution of the CPUs (Column 6 of Table 3), although a non-trivial fraction of the work is useless (Column 7). However, even after producing useful overlap (Column 5), TLS needs to offset significant code bloat (Column 4) to deliver speedups. Finally, while prefetching helps TLS, Figure 6 shows that prefetching's good effect can be overwhelmed by the opposite effects of the TLS overheads.

8.3 Characterization of *TLS4OutOrder*

The remaining columns of Table 3 further characterize *TLS4OutOrder*'s execution. Column 8 shows the frequency of task merges per task commit. We can see that task merge occurs frequently in all codes. On average, there are 0.37 merges per commit. This operation boosts TLS performance, as it increases task size and, as a result, reduces TLS overheads.

Column 9 shows the resulting average number of graduated instructions in the tasks that commit. On average, a task contains 541 instructions.

Finally, the last column shows the percentage of committed dynamic instructions from tasks spawned out of order. It varies noticeably across applications with all codes having a large percentage of dynamic instructions in tasks spawned out-of-order except *bzip2* and *gzip*. Those with a large percentage are the ones responsible for the speedups of *TLS4OutOrder* over *TLS4InOrder* in Figure 6. Generally, the fraction of dy-

[2]In all this discussion, we have only counted graduated instructions. There is an additional waste in both TLS and non-TLS chips caused by misspeculated branches.

App	IPC_{4issue}	IPC_{TLS}	f_{bloat}	$f_{parallel}$	Busy CPUs	Squashed Tasks/ Total Tasks (% Cycles)	# Merges per Task Commit	Task Size (Instr)	Out of Order Dyn. Inst. (%)
bzip2	1.71	2.01	1.06	1.25	1.35	7.4	0.42	744	5.6
crafty	1.50	1.91	1.06	1.45	1.95	25.5	0.28	931	38.6
gap	1.09	1.33	1.04	1.34	1.96	31.5	0.88	1249	88.1
gzip	1.08	1.28	1.07	1.26	1.47	14.1	0.02	626	0.3
mcf	0.04	0.14	1.47	1.69	2.42	30.1	0.20	50	26.3
parser	0.65	0.94	1.22	1.30	1.86	29.8	0.51	165	81.8
twolf	0.78	1.07	1.07	1.57	1.64	4.0	0.28	402	23.7
vortex	1.55	1.97	1.08	1.53	1.80	14.9	0.16	489	77.2
vpr	1.00	1.79	1.27	2.33	3.20	27.2	0.59	212	61.4
Avg	1.05	1.38	1.15	1.53	1.96	20.5	0.37	541	44.8

Table 3: Characterizing the run-time behavior of *TLS4OutOrder*.

namic instructions is correlated with the difference between *TLS4OutOrder* and *TLS4InOrder* in Figure 6: *crafty*, *gap*, *parser*, *twolf*, *vpr*, and *vortex* have high fractions and large differences, while *bzip2* and *gzip* have a small fraction and a small difference. *mcf* is a special case with 26% of the commited dynamic instructions in *mcf* spawned out-of-order and a very large difference. If we consider the dots for *mcf*, we realize that the speed up is largely delivered by the prefetching provided by squashed tasks that were spawned out-of-order. Overall, on average, 45% of the commited dynamic instructions are in tasks spawned out-of-order.

8.4 Architecture Sensitivity Analysis

We have performed other architecture analyses that are not included due to lack of space. Our experiments show that using unlimited size timestamp intervals yields negligible performance gains. Our experiments also show that adding support for dynamic task merging in *TLS4InOrder* barely makes any difference: the average speedup increases by 1%. However, eliminating dynamic task merging from *TLS4OutOrder* causes the average speedup to fall by 29%. This suggests that the potential of our task merging is best exploited when there is more task parallelism, such as in out-of-order spawn environments.

9 Related Work

Out-of-Order Spawning.

Hammond *et al.* [11] propose a TLS CMP where each processor has a co-processor that controls TLS mechanisms with software handlers. They support both subroutine and loop-iteration tasks and, therefore, out-of-order spawn. Co-processors are told what task is running where. They snoop on two broadcast buses and, based on message source, they can tell the relative ordering. Since caches contain state from a single task, no task ID is necessary. Squash signals are also broadcast. Commits require access to a centralized software data structure in shared memory. The most speculative task is killed if there is no space in the CMP. Overall, this is a broadcast-based, relatively centralized architecture. The authors conclude that their scheme has too much software overhead to support subroutine tasks. Their findings motivate our search for hardware-based mechanisms.

There are several high-level performance-evaluation studies of environments that need out-of-order spawn [15, 16, 26, 27]. They often assume an ideal architectural feature, such as an infinite number of processors or perfect value prediction, and compare the performance to more realistic environments. Of those, [15, 16] examine a variety of sources of parallelism, including iterations from multiple loop levels and nested subroutines. [26, 27] examine subroutine-level nested parallelism. None of these papers describe microarchitectural structures to support the tasks used. Consequently, they have not addressed the problems we cover. Our paper is the first microarchitectural design of high-speed out-of-order tasking on a CMP.

DMT is a centralized, SMT-like processor whose hardware can extract out-of-order tasks from unmodified binaries [1]. The design uses centralized structures that are *unusable* in a CMP. Specifically, DMT has a centralized hardware tree that records which tasks are successors of which. To determine the order of two tasks, the hardware walks the tree when: (1) there is a collision in the centralized LD/ST queue, or (ii) a task commits and needs to verify the register predictions for successor tasks. This centralization means that DMT does not need our IS list and timestamp intervals. DMT kills the most speculative task if there is no space in the processor, while we merge tasks to dynamically manage the resources in the system.

Dubey *et al.*'s SPSM [8] is an architecture where tasks are spawned in order. Interestingly, a task can spawn multiple other tasks, but these other tasks cannot further spawn, which guarantees in-order spawn. Our proposed spawning model is more flexible and enables more parallelism.

Littin *et al.*'s WarpEngine [19] is a compute engine where instructions are grouped into 16-instruction branch-less frames. Frames are fetched and executed out of order. The machine appears closer to an aggressive dynamic superscalar that exploits control continuations. For example, it cannot be used as a multiprocessor for parallel applications.

In Multiscalar [20], a task may have multiple exit points. However, only one is correct. Since a task can only spawn a single other *correct* task in its lifetime, Multiscalar supports in-order spawn only.

Related Mechanisms: Timestamping and Merging.

Cleary *et al.* [7] propose several timestamp representations for virtual sequences organized in a tree. They bear some resemblance to our splitting timestamp interval. However, while some of Cleary *et al.*'s schemes are more efficient than others, they all need periodic *re-scaling*. Re-scaling occurs when sequences run out of timestamps. In that case, new timestamps need to be reassigned to *all the tasks* on the fly. This is a very costly operation, which would entail synchronizing the whole machine, and walking all the cache tags, changing all the timestamps. Our splitting timestamp interval scheme is designed for *efficient hardware implementation*. Once a base timestamp is assigned to a task, it never changes. The scheme does not need re-scaling. Thanks to the support for automatic dynamic timestamp expansion (Section 3.1) and timestamp wrap around (Section 4.1), we practically never have to kill a task.

Dubey *et al.*'s SPSM [8] can perform conditional spawns. This is somewhat similar to our dynamic task merging. A key difference is that their mechanism does not allow a parent who initiated a merge to pass the responsibility of completing the merge to a child. Moreover, their mechanism works with in-order spawn only, while ours is for out-of-order spawn, which increases complexity. In addition, in SPSM only the safe task

can perform conditional spawn, while in our mechanism any task can perform task merging. Overall, our mechanism is more flexible. However, it needs a NES counter per task, which may be passed between tasks.

Multiscalar [20] introduces the concept of suppress register. When a task suppresses a section of code (typically a function) the task ignores all the Multiscalar instrumentation in the code. In addition, it increments a counter. Suppressions can be nested, in which case the counter keeps increasing. However, the task *cannot spawn* a successor until all its (nested) suppressed sections are completed and the counter reaches zero. This optimization is typically used to avoid code replication. Our mechanism for dynamic task merging is more flexible. A task can start a merge operation and then, as load conditions change, decide to spawn successors that will complete the merge. This is done by passing the parent's NES counter to its successors. As a result, dynamic task merging is a powerful tool to manage dynamically-changing resources efficiently.

Park *et al.* [17] propose multiplexing a number of in-program-order threads into a single hardware context of IMT. This is very different from our dynamic task merging. In IMT, each of the threads multiplexed in a context still keeps a PC and a rename table pointer. The technique appears similar to recursively applying SMT to each hardware context of an SMT. Our proposal keeps a single PC and a single set of architectural registers for all the merged tasks.

10 Conclusion

This paper has been the first to identify and design a set of microarchitectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. The three mechanisms are Splitting Timestamp Intervals, Immediate Successor List, and Dynamic Task Merging. They address the two main challenges posed by out-of-order spawning: correct and efficient task ordering and resource allocation. With this support and our fully-automated TLS compiler for out-of-order spawn, we unlock the potential of TLS for hard-to-speedup integer codes. Specifically, a TLS CMP with 4 4-issue cores delivers an average speedup of 1.30 for *full* SpecInt 2000 applications; without out-of-order spawn, we obtain an average speedup of 1.04, in line with past TLS CMP work on the same codes. Moreover, our resulting CMP significantly outperforms a 6-issue superscalar, even at the same clock frequency.

These results make TLS a compelling feature, given that they are obtained with a *fully-automated* TLS compiler, on a *decentralized* CMP architecture and, importantly, on *full* SpecInt applications. Moreover, we feel that these results can be improved, as the opportunities for out-of-order spawn in these most challenging applications become better understood.

References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, 1998.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[3] M Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[4] M Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[5] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. In *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 2003.

[6] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[7] J.G. Cleary, J.A.D. McWha, and M.W. Pearson. Timestamp representations for virtual sequences. In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*, 1997.

[8] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, 1995.

[9] SSA for trees - GNU project. URL, May 2003. http://www.gccsummit.org/2003/view_abstract.php?talk=2.

[10] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[12] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[13] X. F. Li, Z. H. Dui, Q. Y. Zhao, and T. F. Ngai. Software Value Prediction for Speculative Parallel Threaded Computations. In *First Value Prediction Workshop*, pages 18–25, June 2003.

[14] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 365–372, June 1999.

[15] P. Marcuello and A. Gonzalez. A Quantitative Assessment of Thread-level Speculation Techniques. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 595–604, 2000.

[16] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[17] I. Park, B. Falsafi, and T. Vijaykumar. Implicitly Multithreaded Processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[18] J. Renau. *Chip Multiprocessors with Speculative Multithreading: Design for Performance and Energy Efficiency*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[19] R.H.Littin, J.A.D. McWha, M.W.Pearson, and J.G.Cleary. Block based execution and task level parallelism. In *Australian Computer Science Communications*, pages 57–66, 1998.

[20] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, June 1995.

[21] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[22] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[23] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[24] J. Y. Tsai, Z. Jiang, and P. C. Yew. Compiler Techniques for the Superthreaded Architecture. In *International Journal of Parallel Programming*, pages 27(1):1–19, 1999.

[25] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[26] F. Warg and P.Stenström. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[27] F. Warg and P. Stenström. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[28] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[29] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 65–77, November 2002.